FRESH: Towards Efficient Graph Queries in an Outsourced Graph

Kai Huang[¶,†,§], Yunqi Li[†,§], Qingqing Ye[‡], Yao Tian[†], Xi Zhao[†], Yue Cui[†], Haibo Hu[‡], Xiaofang Zhou[†]

[¶] Faculty of Information Technology, Macau University of Science and Technology, Macau SAR
[†] The Hong Kong University of Science and Technology, Hong Kong SAR
[‡] The Hong Kong Polytechnic University, Hong Kong SAR
huangkai@must.edu.mo, ylilo@connect.ust.hk, qqing.ye|haibo.hu@polyu.edu.hk, ytianbc|xzhaoca|ycuias|zxf@ust.hk

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00346

***Abstract*—The constantly increasing scale of graphs leads to higher costs in terms of data storage and computation. Consequently, there is a growing trend of outsourcing and analyzing graphs in clouds. As there is a concern that cloud servers may extract sensitive information from these graphs, the graphs being outsourced must be pre-anonymized, leading to increased space consumption and degraded graph query processing efficiency. Previous work has attempted to address this issue by outsourcing a compacted anonymized graph to the cloud. However, the solution typically focuses on a specific type of query, such as a subgraph query, and cannot adequately accommodate real-life scenarios where multiple applications often work concurrently on the same graph. In this paper, we propose a generic framework called FRESH to handle various graph queries efficiently within a single outsourced graph. To reduce the size of the outsourced graph, we developed a novel graph contraction scheme that transforms a big graph into a compact one while preserving graph privacy. To showcase the adaptability of classical graph query algorithms (e.g., subgraph query, triangle counting, and shortest distance query), we demonstrate their successful execution on the same compact graph created through our contraction scheme. We further extend our framework by incorporating optimizations that significantly improve query processing efficiency. Extensive experimental results demonstrate the superiority of FRESH over traditional techniques.**

***Index Terms*—Graph Queries, Outsourced Graph, Efficiency**

I. Introduction

Graphs have become increasingly popular for modeling complex and interconnected data and have been widely applied in social networks, bioinformatics, and computer vision. However, with the increasing size of graphs, storing and processing the graph data can impose expensive upfront infrastructure costs on users, such as start-up companies, which hinders the acquisition of valuable information from graphs. To address this problem, many cloud service providers (e.g., Amazon, Alibaba, and Microsoft Azure) offer graph outsourcing services by storing user-owned graphs and performing query processing tasks on their behalf. However, the cloud server may not be fully trusted, and is generally described as "honest but curious" or "semi-honest" [2]–[4]. On the one hand, the cloud server behaves honestly by adhering to the designated protocol specification (*e.g.,* HIPAA compliance[1]) and correctly computing queries. However, on the other hand, the server may be curious about the privacy of the graph data. A main form of privacy disclosure is identity (*also known as* structural information) disclosure, which compromises the location of a target node in the graph and can be caused by various structural attacks, such as degree attacks, hub-fingerprint attacks, 1-neighbor-graph attacks, and subgraph attacks [13]–[15]. To defend against these attacks, numerous privacy-preserving methods [5], [6], [9] have been proposed to enforce symmetry in an outsourced graph by introducing noise edges and nodes. This reduces the probability of a target being identified by attackers to at most $1/k$. The k-automorphism model [5], in particular, is such a method that transforms a graph G into a k-automorphic graph G^k, where each subgraph (resp. vertex) has at least $(k-1)$ other symmetric subgraphs (resp. vertices). Put another way, the attacker cannot differentiate between a vertex and the other $(k-1)$ symmetric vertices. It is known that the k-automorphism strategy can defend against any known structure-based attack [4], [5].

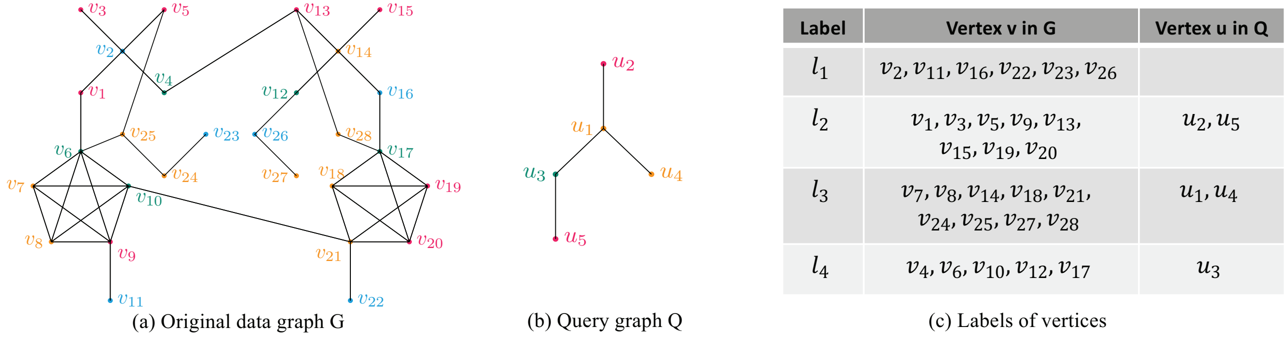
Example 1: Consider the graph G shown in Figure 1(a), where each vertex represents an entity and its label is displayed in Figure 1(c). If an adversary knows that an entity has 6 neighbors, they can immediately deduce that vertex v_6 in G is the target entity (i.e., degree attack). Additionally, if the adversary has prior knowledge of the relationships of an entity u_1 and its neighbors, for example, Q in Figure 1(b), they can locate u_1 with a subgraph attack. Specifically, by checking subgraph isomorphism (as defined in Definition 1) from Q to G, the adversary can identify the matched subgraph of Q in G, *i.e.,* $\langle v_5, v_{25}, v_6, v_{24}, v_9\rangle$, where v_5, v_{25}, v_6, v_{24}, and v_9 match with u_2, u_1, u_3, u_4, and u_5, respectively. Therefore, they can locate u_1 in G, *i.e.,* v_{25}.

The k-automorphism model can be utilized to prevent such structure attacks by introducing noise edges to construct a k-automorphic graph G^k. For instance, G^k in Figure 2(a) is a k-automorphic graph of G, where $k = 2$. In this figure, the noise edges (*e.g.,* edge (v_4, v_{23})) are displayed by black dashed lines. Notably, each vertex in Block 0 (as seen in the dashed box) has a symmetric vertex in Block 1, and their labels are now the same, *i.e.,* the union of their labels, as shown in the Alignment Vertex Table (AVT) in Figure 2(b). v_1 in Block 0 (*i.e.,* B_0)

[§]These authors contributed equally to this work.
[1]https://aws.amazon.com/compliance/hipaa-compliance/

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00346

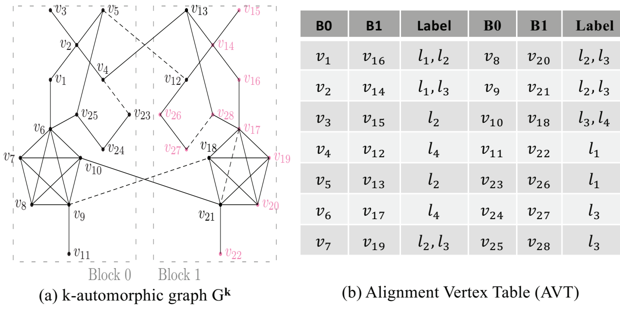
Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.



(a) Original data graph G (b) Query graph Q (c) Labels of vertices

Label	Vertex v in G	Vertex u in Q
l_1	$v_2, v_{11}, v_{16}, v_{22}, v_{23}, v_{26}$	
l_2	$v_1, v_3, v_5, v_9, v_{13}, v_{15}, v_{19}, v_{20}$	u_2, u_5
l_3	$v_7, v_8, v_{14}, v_{18}, v_{21}, v_{24}, v_{25}, v_{27}, v_{28}$	u_1, u_4
l_4	$v_4, v_6, v_{10}, v_{12}, v_{17}$	u_3

Fig. 1: A sample data graph and query graph



B0	B1	Label	B0	B1	Label
v_1	v_{16}	l_1, l_2	v_8	v_{20}	l_2, l_3
v_2	v_{14}	l_1, l_3	v_9	v_{21}	l_2, l_3
v_3	v_{15}	l_2	v_{10}	v_{18}	l_3, l_4
v_4	v_{12}	l_4	v_{11}	v_{22}	l_1
v_5	v_{13}	l_2	v_{23}	v_{26}	l_1
v_6	v_{17}	l_4	v_{24}	v_{27}	l_3
v_7	v_{19}	l_2, l_3	v_{25}	v_{28}	l_3

(a) k-automorphic graph G^k (b) Alignment Vertex Table (AVT)

Fig. 2: k-automorphic graph and alignment vertex table

is symmetric to v_{16} in Block 1 (*i.e.,* B_1), and their label is now $\{\ell_1, \ell_2\}$ (see the first row in Figure 2(b)). Consequently, the probability that the adversary locates v_6 with a degree attack is at most $\frac{1}{2}$, as there is at least one vertex (*i.e.,* v_{17}, the symmetric vertex of v_6) that has the same degree as v_6. Similarly, the probability that the adversary locates u_1 in G with the same subgraph attack is at most $\frac{1}{2}$, as there are at least two matched subgraphs of Q on G. ∎

However, outsourcing a k-automorphic graph to a cloud and processing graph queries on it has some significant limitations. Firstly, while enforcing symmetry in an outsourced graph can defend against structural attacks, it unavoidably enlarges the size of a graph and increases space consumption, which further degrades query processing efficiency. Secondly, existing techniques (*e.g.,* [2], [4]) that outsource only a fraction of a graph into the cloud for reducing storage space target specific type of queries (*e.g.,* subgraph query), and cannot adequately accommodate real-life scenario where multiple applications often run on the same graph concurrently [2].

To address these limitations, we propose a generic framework called FRESH (e**F**ficient g**R**aph queri**E**s in out**S**ourced grap**H**s) to handle various graph queries efficiently on the same outsourced graph. To reduce the size of the outsourced graph, we develop a novel graph contraction scheme to transform a big graph into a compact one while preserving the graph privacy. In particular, we first adopt the k-automorphism model to anonymize the original graph for privacy concerns. As this anonymization will inevitably enlarge the size of a graph, we further take advantage of the symmetric property of a k-automorphic graph and graph contraction to generate a compact graph for outsourcing to a cloud. Once the compact graph is outsourced to the cloud, as a proof of concept, we adapt three classical graph query algorithms (*i.e.,* subgraph query, triangle counting, and the shortest distance query) to the outsourced graph. We selected these query classes based on the dichotomies [18]: (1) label-based (subgraph query) vs. non-label-based (triangle counting, shortest distance query); (2) local (subgraph query, triangle counting) vs. non-local (shortest distance query). To efficiently process such queries, we design novel optimizations to further boost query efficiency. Extensive experimental results demonstrate the superiority of FRESH to traditional techniques.

To summarize, we make the following contributions: (i) we propose a generic framework called FRESH to handle various graph queries efficiently in the same outsourced graph. In the framework, a novel and lossless graph contraction scheme is designed to reduce the size of outsourced graphs while preserving private structure information. (ii) we adapt three classical graph query algorithms (*i.e.,* subgraph query, triangle counting, and the shortest distance query) to the compacted graphs to show that existing graph queries can be easily adapted to our contraction scheme. (iii) we extend our framework by incorporating optimization strategies to boost query efficiency without compromising graph privacy. (iv) we conduct extensive experimental evaluations to show the superiority of our methods over existing solutions.

The rest of this paper is organized as follows. Section II discusses related work. Section III presents the preliminaries. In Section IV, we introduce our FRESH framework and the novel contraction scheme. Three adapted graph queries are presented in Section V. We present post-processing in Section VI, optimizations in Section VII, and experimental results in Section VIII. We conclude the work in Section IX. Formal proofs of lemmas and theorems are in [1].

[2] As reported in a study on GDB benchmarks [17], an average of 10 classes of queries are executed concurrently on a single graph.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

II. Related Work

Privacy-preserving graph data publication and anonymization. A variety of structural privacy-preserving mechanisms such as [5], [6], [9] have been developed to defend against various structural attacks [13]–[15] such as degree attacks, hub fingerprint attacks and subgraph attacks, by enforcing symmetry in an outsourced graph. In particular, given a graph G, k-automorphism model developed by Zou et al. [5] transforms G into a k-automorphic graph G^k by introducing some noise edges and vertices, where each vertex has at least $(k-1)$ other symmetric vertices. Hence, there are no structural differences between any vertex and its $(k-1)$ symmetric vertices. Recently, data privacy and security are becoming more and more important [10], [53], [60]–[63]. Classical anonymization methods such as k-anonymity [10], ℓ-diversity [11], t-closeness [12] and differential privacy [2], [44]–[47], [50]–[54] can also be adapted for protecting sensitive labels (*e.g.,* salary, social security number, and medical history of a user) of graphs. Compared to this line of research that only focuses on privacy-preserving graph data publication and anonymization, this paper is mainly on efficient graph queries in outsourced graphs.

Graph queries and privacy-preserving graph queries in a cloud. There has been a host of work on graph queries such as the subgraph query [19]–[25], [40], [41], [48], [49], [56], [57], triangle counting [26], [27] and the shortest distance/path query [30]–[34], [58], [59]. Recent efforts on privacy-preserving methods or frameworks have been made for these graph queries (*e.g.,* [35]–[39]). In particular, [35] considers secure shortest path queries by anonymizing edge weights in the original graph while preserving shortest paths. [36] discusses privacy-preserving shortest distance queries in a cloud. [37] presents a novel technique for answering subgraph queries over encrypted graphs for a graph database consisting of a set of small graphs. [4], [42] present privacy-preserving subgraph matching methods on a large graph. Unfortunately, they support only one query type. In practice, multiple applications often run on the same graph simultaneously [17], [18].

Graph contraction-related methods. Graph compression and summarization have been extensively studied using various techniques. Graph compression techniques [55], [64] aim to create query-specific equivalence relations by merging equivalent vertices into a single vertex. On the other hand, graph summarization [65], [66] generates an abstraction or summary of the original graph by aggregating nodes or subgraphs to enhance query efficiency. However, this summarization may result in some loss of information about the original graph. In contrast, graph contraction [18] reduces a large graph into smaller ones by contracting subgraphs, which is lossless and can retrieve precise answers for queries. In this paper, we propose a novel privacy-preserving graph contraction that ensures the resulting compact graph can withstand any structural attacks. This contraction scheme is distinct from existing indexing techniques [33], [71] such as indexing for subgraph queries [67], [68] and the shortest distance queries [69], [70], each of which is designed for a particular query class.

TABLE I: List of key notations.

Notation	Description
G (or g), G^k, G^k_{small}, G^{out}	a graph, a k-automorphic graph, a small fraction of G^k, an outsourced graph
$R^{\mathcal{QS}}$	a set of answer to multiple query class $\mathcal{QS}$
$V(G)$, $E(G)$, $l_G(\cdot)$	vertex set, edge set and label function of G
$w(u,v)$	weight of the edge (u,v)
$\mathcal{Q}$	a query class
$\mathcal{QS}$	multiple query classes, *i.e.,* $\mathcal{Q} \in \mathcal{QS}$
Q	a query *i.e.,* $Q \in \mathcal{Q}$
c_l, c_u	minimum or maximum size of a supernode
p, $\mathcal{P}$	a path, a set of paths
C, $C(u)$	candidate set for vertices in Q and $u \in Q$
π	matching order for subgraph isomorphism
$\mathcal{M}$	a set of embeddings of query Q in graph G
$\langle f_A, f_C, \mathcal{S}, f_D \rangle$	<mapping function, contraction function, synopses, decontraction function>
f'_C	reverse function of f_C
V_{B_0}, V_{N_0}	vertices in the first block of G^k, one-hop neighbors of vertices in V_{B_0}
CBV	supernode consisting of vertices in V_{N_0}
V_H, E_H	supernodes and superedges of subgraph H
$d(u,v)$	the shortest distance between u and v

III. Preliminaries

Table I lists key notations and acronyms used in this paper.

A. Key Concepts

Graphs and (Sub)graph Isomorphism. We consider an undirected graph $G = (V, E, L)$ where V is the vertex set, $E \subseteq V \times V$ is the edge set and L is a label function such that $L(v)$ is the label of $v \in V$. Each edge $e \in E$ between two vertices u and v is denoted by (u,v) and associated with a positive weight $w(u,v)$.

*Definition 1 (**Subgraph Isomorphism**):* Given graphs g and G, a subgraph isomorphism from g to G is an injection function f: $V(g) \rightarrow V(G)$ such that 1) $\forall v \in V(g)$, $l_g(v) = l_G(f(v))$ and 2) $\forall (u,v) \in E(g)$, $(f(u), f(v)) \in E(G)$ and $l(u,v) = l_G(f(u), f(v))$ where l_g and l_G are the labeling functions of g and G, respectively.

We say g is *subgraph isomorphic to* G (denoted by $g \subseteq G$) if there exists at least one subgraph isomorphism from g to G. We also say that g is *isomorphic to* G if $g \subseteq G$ and $G \subseteq g$.

Graph Automorphism. A graph automorphism of a graph is a form of symmetry in which the graph is mapped onto itself while preserving the edge–vertex connectivity. Formally,

*Definition 2 (**Graph Automorphism [5]**):* An automorphism of a graph G is an automorphic function f of the vertex set $V(G)$, such that for any edge $e = (u,v)$, $f(e) = (f(u), f(v))$ is also an edge in G and $l_G(u)$ (resp. $l_G(v)$) $= l_G(f(u))$ (resp. $l_G(f(v))$) where l_G is the labeling function of G.

Figure 3 illustrates six automorphisms, such as $(1,2,3,4)$ in (a) and $(2,1,3,4)$ in (b), of a star graph with a size of 4, where the center node is connected to 3 neighbors. Let's consider Figure 3(a) and (b). In this case, there exists an automorphic function f such that $f(1) = 2, f(4) = 4$, and $f(e = (1,4)) =$

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

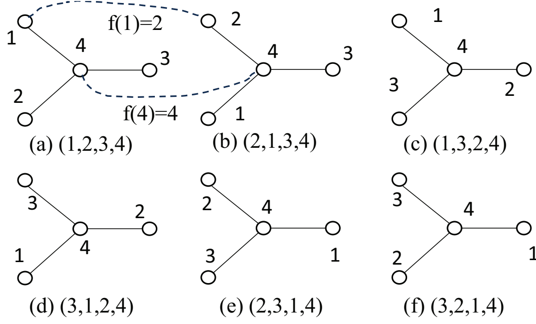


Fig. 3: Graph Automorphism

$(2, 4)$. A graph G is called *k-automorphic graph* if there exist k graph automorphisms in G.

Example 2: Figure 2(a) and Figure 2(b) depict G^k, k-automorphic graph ($k = 2$) of G (see Figure 1), and Alignment Vertex Table (AVT, *i.e.,* symmetric relations of vertices in G^k), respectively. Observe that graph automorphism brings several noise edges (dashed lines in G^k) to form an automorphic structure. Also, for a group of k symmetric vertices in AVT, they hold the same label set which is the combination of all their labels in original data graph G. ∎

It is known that k-automorphic graph can defend against any structure-based attack [5] such as degree attack, hub-fingerprint attack, 1-neighbor-graph attack and subgraph attack. Given a graph G, K-MATCH algorithm [5] can transform it into a k-automorphic graph.

Graph Queries. A graph query Q is a computable function from a graph G to another object such as a real value, graph, and relation. As a proof of concept, we consider three query classes: subgraph query, triangle counting and the shortest distance query.

- *Subgraph (Matching) Query (SubA):* Given a data graph G and a query graph g, a subgraph query Q aims to find a set of subgraphs in G that are isomorphic to g. Formally, $Q(g, G) = \{g_i\}$ where g_i is a subgraph of G and isomorphic to g.
- *The Shortest Distance Query (Dist):* Given a data graph G and a vertex pair $\langle o, d\rangle$, the shortest distance (*also known as* point-to-point shortest distance) query finds the shortest distance between o and d, *i.e.,* $Q(o, d, G) = argmin_{p\in\mathcal{P}}\sum_{(u,v)\in p} w(u, v)$ where $\mathcal{P}$ is the set of all possible paths between o and d.
- *Triangle Counting (TriC):* Given a data graph G, a triangle counting query Q is to find the number of triangles in G. That is, $Q(G) = |TriC(G)|$ where $|TriC(G)|$ is the number of triangles in G.

The shortest distance query between two vertices u and v is considered a non-local query because there is no fixed distance d independent of the input graph such that $d(u, v) < d$. On the other hand, both the subgraph query and triangle counting are classified as local queries, meaning they exhibit locality. This is because a single subgraph match or triangle is confined to a specific local area within the input graph G.

B. Problem Statement

Observe that multiple real-life query classes (*i.e.,* applications) $\mathcal{QS}$ often run on the same graph G at the same time [17]. A query class $\mathcal{Q} \in \mathcal{QS}$ contains a set of graph queries of the same query type (*e.g.,* subgraph query) and each graph query $Q \in \mathcal{Q}$ is a computable function discussed above. Due to the high cost of graph storage and computation, G can be outsourced and analyzed in a cloud, which may be curious to infer the sensitive information of G. Therefore, it is an important problem to process graph queries in outsourced graphs without disclosing graph privacy. Given a graph G and multiple query classes $\mathcal{QS}$, the task of graph queries in an outsourced graph involves two main subtasks,

- *graph outsourcing:* transforming G into a graph G^{out} ($G \subseteq G^{out}$) such that G^{out} can defend against any structure-based attack, and outsourcing G^{out} to a cloud;
- *query processing:* processing multiple query classes $\mathcal{QS}$ in the same outsourced graph G^{out} and returning the results to the client.

Remark. Note that G is a subgraph of the outsourced graph G^{out} such that the *exact* query results on G^{out} can be covered by that on G. In addition, graph outsourcing is performed on the client side while query processing runs on a cloud. When the cloud returns the query results to the client side, only lightweight filtering computations are needed. In particular, the client only needs to filter the results that contain the parts (*e.g.,* edges) not in G.

C. Baselines

We propose three baseline methods for addressing this problem. The first baseline, called AUT, involves using the K-MATCH algorithm [5] to transform a given graph G into a k-automorphic graph G^k. The transformed graph G^k is then outsourced to a cloud for query processing. The second baseline, named SUC, also utilizes the K-MATCH algorithm to transform graph G into G^k. However, it takes a different approach by following existing work [2], [4] to outsource a succinct version of G^k. The third baseline, called CAU, applies the contraction method [18] directly on the k-automorphic graph G^k. However, as demonstrated in Section VIII, these baseline methods have their shortcomings. They suffer from high storage costs and low query performance.

IV. FRESH: A Novel Framework

In this section, we begin by identifying the design challenges associated with the problem. Subsequently, we introduce our FRESH framework to address these challenges.

A. Design Challenges

The problem of graph queries in outsourced graphs introduces nontrivial challenges. First, outsourcing a k-automorphic graph to a cloud unavoidably enlarges the size of a graph and increases space consumption, which further degrades query processing efficiency. Second, directly applying the existing

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

contraction scheme [18] to reduce the size of a k-automorphic graph does not consider the symmetric property of the graph, potentially resulting in higher space requirements and lower computational efficiency (refer to Section VIII). Additionally, alternative graph compression and summarization techniques may cause information loss regarding the original graph (refer to Section II).

B. Overview of FRESH

To address these challenges, we propose the FRESH framework to efficiently process multiple classes of graph queries in an outsourced graph, which can preserve the privacy of the graph. The main idea of FRESH is to adopt the k-automorphism model to anonymize the original graph for privacy concerns, and take advantage of the symmetric property of the k-automorphic graph and graph contraction to generate a compact graph, which supports classical graph query algorithms and facilitates query processing.

FRESH is outlined in Algorithm 1. Formally, given the original graph G and query classes $\mathcal{QS}$, FRESH aims to find a set of results, $R^{\mathcal{QS}}$, for all queries in $\mathcal{QS}$. Specifically, it first initializes $R^{\mathcal{QS}}$ with an empty set ϕ (Line 1, Algorithm 1). Then, it adopts the following two steps to derive the privacy-preserving graph contraction scheme $\langle f_A, f_C, \mathcal{S}, f_D \rangle$ and compute the contracted graph G^{out}: (1) pre-processing the original graph to generate f_C, f_A, f_D (PREPROCESSGRAPH, Line 2); and (2) contracting graph to derive $\mathcal{S}$ and G^{out} (CONTRACTGRAPH, Line 3). In particular, f_A is a mapping function, which maps a graph G to a small fraction of k-automorphic graph G^k (denoted by G^k_{small}, Definition 3); f_C is a contraction function, which takes G^k_{small} as input and produces a compact graph G^{out} by collapsing certain subgraphs H into supernodes v_H; $\mathcal{S}$ is synopses, which annotates each supernode v_H of G^{out} with a synopsis $S(v_H) \in \mathcal{S}$; f_D is a decontraction function, which restores each supernode v_H in G^{out} to its original subgraph H. Once the cloud side receives a query class $\mathcal{Q} \in \mathcal{QS}$, it adopts the ANSWERQUERIES algorithm (see Section V) to process queries on G^{out} with the help of $\mathcal{S}$, f_C, f_D, and f_A (Line 5). Finally, the processed results are returned to the client side where these results are further refined by PROCESSRESULT Procedure (Line 6) based on the original graph G and the graph contraction scheme. In a nutshell, FRESH offers several advantages:

(i) Limited memory consumption. By taking advantage of the symmetric property of the k-automorphic graph, FRESH presents a privacy-preserving graph contraction to avoid excessive memory consumption for storing an entire enlarged graph on which query processing is performed. Compared to the existing graph contraction [18], the privacy-preserving graph contraction introduces a new special supernode called CBV (see Section IV-C) and the mapping function f_A.

(ii) Effective query processing. FRESH adapts three classical graph query algorithms (*i.e.,* subgraph query, triangle counting, and the shortest distance query) to the contracted graph, and offers efficient query performance.

(iii) Guaranteed query results. Both the graph contraction and query processing stages in FRESH are lossless, meaning that the integrity of the query results is preserved. Additionally, post-processing techniques can be applied to prune false positive results, further enhancing the accuracy of the query results. Consequently, FRESH guarantees exact query results.

In what follows, we present the details of the privacy-preserving graph contraction scheme including preprocessing (*i.e.,* PREPROCESSGRAPH) in Section IV-C and graph contraction (*i.e.,* CONTRACTGRAPH) in Section IV-D. In Section V, we discuss how classical graph queries are adapted to the outsourced graph (*i.e.,* ANSWERQUERIES). The post-processing algorithm (*i.e.,* PROCESSRESULT) is provided in Section VI.

Algorithm 1 FRESH

Input: Original graph G, multiple query classes $\mathcal{QS}$.
Output: A set of results $R^{\mathcal{QS}}$ of queries.

1: $R^{\mathcal{QS}} \leftarrow \phi$
2: $f_A, G^k_{small}, f_C, f_D \leftarrow$ PREPROCESSGRAPH(G)
3: $G^{out}, \mathcal{S} \leftarrow$ CONTRACTGRAPH(G^k_{small}, f_C)
4: **for each** $\mathcal{Q} \in \mathcal{QS}$ **do**
5: $\quad R^{\mathcal{Q}} \leftarrow$ ANSWERQUERIES(G^{out}, $\mathcal{Q}$, f_A, f_C, f_D, $\mathcal{S}$)
6: $\quad R^{\mathcal{Q}}_G \leftarrow$ PROCESSRESULT (G, $R^{\mathcal{Q}}$, f_A)
7: $\quad R^{\mathcal{QS}} \leftarrow R^{\mathcal{QS}} \cup R^{\mathcal{Q}}_G$
8: **return** $R^{\mathcal{QS}}$

C. Pre-Processing on Original Graph

FRESH processes the original graph G to derive a mapping function f_A, contraction function f_C, decontraction function f_D and a small fraction of G^k, *i.e.,* G^k_{small} below.

Definition 3: Given k-automorphic graph G^k, vertices in the first block of G^k is denoted by V_{B_0}. The one-hop neighbors of vertices in V_{B_0} are denoted by V_{N_0}. G^k_{small} is a subgraph of G^k and $G^k_{small} = (V_{small}, E_{small})$, where $V_{small} = V_{B_0} \cup V_{N_0}$, $E_{small} \subseteq V_{small} \times V_{small}$.

Computing f_A and G^k_{small}. Given a graph G, it first transforms G into a k-automorphic graph G^k by the K-MATCH algorithm. Then, it constructs G^k_{small} by selecting V_{B_0}, V_{N_0} and edges between any two vertices in $V_{B_0} \cup V_{N0}$. Finally, for each vertex v in G^k_{small}, f_A stores its block id (*i.e.,* $part$, Figure 5(c)), symmetric vertices (*i.e.,* $v._{sym}$), and the union set (denoted by $label$, Figure 5(c)) of its label and labels of symmetric vertices. When constructing the k-automorphic graph, it is possible to introduce some noise edges, which further introduce false shortest distances. We address this problem by assigning a noise edge (u, v) with the weight $w(u, v) = min_{t \in N(u,v)}(w(u, t) + w(t, v)) + \epsilon$ where $N(u, v)$ is the common neighbors[3] of u and v, and $\epsilon > 0$ is drawn from a Laplace distribution.

Example 3: Consider the noisy edge (u, v) in Figure 4. The common neighbors of u and v include t_1 and t_2. Since $w(u, t_1) + w(t_1, v) = 1.3 + 1.5 = 2.8$ and $w(u, t_2) + w(t_2, v) = 1.4 + 1.1 = 2.5$, the noise edge (u, v) is assigned a weight $w(u, v) = min_{t \in N(u,v)}(w(u, t) + w(t, v)) + \epsilon = (1.4 + 1.1) +$

[3]If $N(u, v) = \emptyset$, $w(u, v) = min_{p \in P(u,v)} dis(p) + \epsilon$ where $P(u, v)$ is all paths between u and v, and $dis(p)$ is the distance of a path p.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

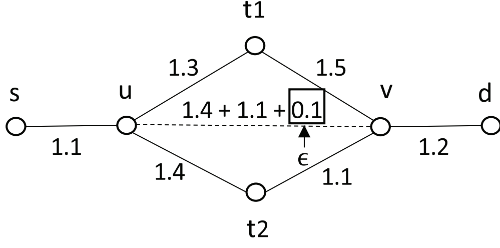


Fig. 4: Noise Edge (u, v)

$0.1 = 2.6$ where $\epsilon = 0.1$ is Laplace noise. Note that despite the existence of the noise edge, the shortest distance between s and d is preserved as 4.8. ∎

Computing f_C and f_D. Given G^k_{small}, FRESH initially contracts all nodes in V_{N_0} into a single supernode. We refer to this supernode, which consists of all nodes in V_{N_0}, as CBV. Next, it proceeds to contract $[c_l, c_u]$ vertices in V_{B_0} into supernodes, following the order of clique, star, and path contractions. Specifically, (1) for cliques, it iteratively selects an uncontracted node that is connected to all previously selected nodes until the clique is fully contracted; (2) for stars, it first selects a central node and then repeatedly chooses an uncontracted leaf node connected to the center and disconnected from all previously selected leaves; (3) for paths, it identifies intermediate nodes that have only two neighbors and the corresponding neighbors (or neighbors of neighbors) that are disconnected. Finally, the contraction function f_C is used to map each node in G^k_{small} to a supernode. Furthermore, f'_C serves as the reverse function of f_C. A decontraction function f_D restores the subgraph (*i.e.,* all edges between nodes in a supernode) contracted to a supernode and edges between two nodes located in different supernodes. In particular, for a supernode $v_H, f_D(v_H)$ restores the edges between the nodes in $f'_C(v_H)$. For a superedge $(v_{H_1}, v_{H_2}), f_D(v_{H1}, v_{H2})$ restores the edges between $f'_C(v_{H1})$ and $f'_C(v_{H2})$ and the corresponding edge weights.

Example 4: Reconsider Example 5. The decontraction function f_D can restore the subgraph in Figure 5(a) from supernodes in Figure 5(b), *e.g.,* $f_D(v_{H_1})$ is a star with central node v_2 and leaves v_1, v_3 and v_5; as well as edges from superedges, *e.g.,* $f_D(v_{H_1}, v_{H_3}) = \{(v_1, v_6)\}$. ∎

D. Graph Contraction

Contracted Graph G^{out}. Based on f_C, the contracted graph is represented as $G^{out} = f_C(G^k_{small}) = (V_H, E_H, f'_C)$, where:

- V_H is a set of supernodes. Each supernode is mapped from a subgraph H in G^k_{small}.
- $E_H \subseteq V_H \times V_H$ is a set of superedges. Given two supernodes v_{H_1} and v_{H_2}, there is a superedge $(v_{H_1}, v_{H_2}) \in E_H$ if there exist nodes v_1 and v_2 such that $f_H(v_1) = v_{H_1}, f_H(v_2) = v_{H_2}$ and $(v_1, v_2) \in E$;
- f'_C is a reverse function of f_C. $f'_C(v_H) = \{(v, L(v)) \mid f_C(v) = v_H\}$ where $L(v)$ is labels of v.

Example 5: Consider the graph G^k_{small} in Figure 5(a), the contraction function f_C contracts G^k_{small} to the contracted graph G^{out} in Figure 5(b) such that $G^{out} = f_C(G^k_{small}) = (V_H, E_H, f'_C)$. Here, the supernode set $V_H = \{v_{H_1}, v_{H_2}, v_{H_3}, v_{H_4}, v_{H_5}\}$; the superedge set $E_H = \{(v_{H_1}, v_{H_2}), (v_{H_2}, v_{H_3}), (v_{H_3}, v_{H_4}), (v_{H_3}, v_{H_5}), ...\}$; the reverse function $f'_C(H_3)$ finds the labels for each node in the subgraph H_3, *i.e.,* $f'_C(H_3)$=$\{(v_6, \{l_4\}), (v_7, \{l_2, l_3\}), (v_8, \{l_2, l_3\}), (v_9, \{l_2, l_3\}), (v_{10}, \{l_3, l_4\})\}$. ∎

Synopses $\mathcal{S}$. Each supernode v_H in the contracted graph G^{out} carries a synopsis $\mathcal{S}(v_H) \in \mathcal{S}$ for each query class $\mathcal{Q}$. Therefore, there are three synopses $\mathcal{S}$, $\mathcal{S}_{SubA}$, $\mathcal{S}_{Dist}$, and $\mathcal{S}_{Tric}$. In particular, if

(1) $\mathcal{Q} = SubA$, $\mathcal{S} = \mathcal{S}_{SubA}$: the synopsis of v_H consists of the node type (*i.e., $type$*, see Figure 5(d)) and the auxiliary information (*i.e.,* $Info$, Figure 5(d)).

- v_H.*type*: the type of supernodes consists of path, clique, star, CBV, and singleton.
- v_H.*Info*: v_H.*Info* = v_H.*list*= $\langle v_1, v_2, ..., v_x \rangle$ where $(v_i, v_{i+1}) \in E$ and x is the length of the path, if v_H.*type*= $path$; v_H.*Info* = v_H.*vset*= $\langle v_1, v_2, ..., v_x \rangle$ where v_i is a vertex in v_H, and x is the number of vertices in v_H, if v_H.*type*= CBV; v_H.*Info* = v_H.*c*, *i.e.,* the center node of a star, if v_H.*type* = $star$.

(2) $\mathcal{Q} = Dist$, $\mathcal{S} = \mathcal{S}_{Dist}$: The synopsis $\mathcal{S}_{Dist}$ of v_H is an extension of $\mathcal{S}_{SubA}$, which includes the auxiliary information on distance (*i.e., v_H.dis*). For a supernode v_H, v_H.*dis* is *i.e.,* a triple $(v_1, v_2, d_{v_H}(v_1, v_2))$ for a path between v_1 and v_2 in v_H. If v_H.*type* = CBV, v_H also keeps track of the distances between vertices $v_1 \in v_H$ and $v_2 \in CBV$.

Example 6: Consider the contracted graph G^{out} in Figure 5(b). If $w(u, v) = 1$ for all edges (u, v), $v_{H1}.dis = \{(v_3, v_2, 1), (v_1, v_2, 1), (v_5, v_2, 1), (v_3, v_1, 2), (v_3, v_5, 2), (v_1, v_5, 2)\}$. $v_{H4}.dis = \{(v_4, v_{23}, 1), (v_{23}, v_{24}, 1), (v_{24}, v_{25}, 1), (v_4, v_{24}, 2), (v_4, v_{25}, 3), (v_{23}, v_{25}, 2)\}$. ∎

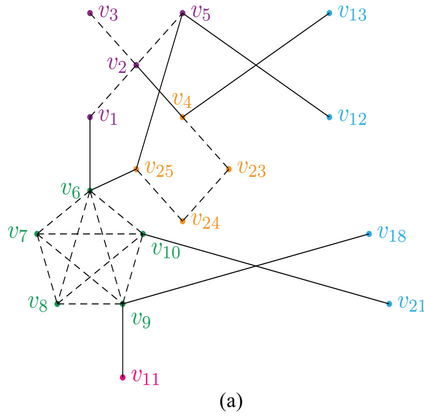
(3) $\mathcal{Q} = Tric$, $\mathcal{S} = \mathcal{S}_{Tric}$: The synopsis $\mathcal{S}_{Tric}$ of v_H is an extension of $\mathcal{S}_{SubA}$, which includes the auxiliary information on triangles (*i.e., v_H.tc*). v_H.*tc* is a set of triangles that cross at least two different supernodes including v_H. If v_H.*type* = CBV, v_H.*tc* includes triangles in CBV.

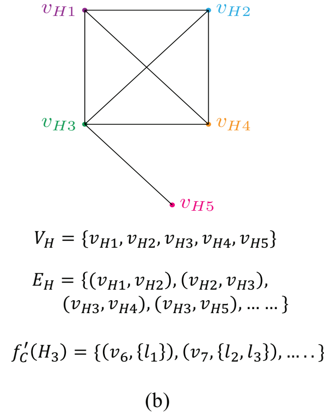
Lemma 1: The time complexities of k-automorphic graph construction and graph contraction are $\mathcal{O}(\sum_{i \leq k} |E(B_i)|)$ and $\mathcal{O}((|V(G^k)| + |E(G^k)|)^2)$, respectively, where $|E(B_i)|$ is the number of vertices in the block B_i of G^k and $|V(G^k)|$ (resp. $|E(G^k)|$) is the number of vertices (resp. edges) in G^k.

V. Query Processing in a Cloud

In this section, we demonstrate how existing graph query processing algorithms can be easily adapted to outsourced graphs G^{out} to facilitate query processing including subgraph isomorphism, shortest distance, and triangle counting. When a query $Q \in \mathcal{Q}$ is received, FRESH first checks whether the synopsis on a supernode v_H contains enough information to answer Q. If so, FRESH answers the query with v_H without further processing. Otherwise, FRESH decontracts v_H by the decontraction function f_D. The synopsis of v_H often

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.





(c)

vertex	part	*Info*		vertex	part	*Info*	
		$v_{.sym}$	label			$v_{.sym}$	label
v_1	B_0	v_{16}	l_1, l_2	v_{10}	B_0	v_{18}	l_3, l_4
v_2	B_0	v_{14}	l_1, l_3	v_{11}	B_0	v_{22}	l_1
v_3	B_0	v_{15}	l_2	v_{23}	B_0	v_{26}	l_1
v_4	B_0	v_{12}	l_4	v_{24}	B_0	v_{27}	l_3
v_5	B_0	v_{13}	l_2	v_{25}	B_0	v_{28}	l_3
v_6	B_0	v_{17}	l_4	v_{12}	B_1	v_4	l_4
v_7	B_0	v_{19}	l_2, l_3	v_{13}	B_1	v_5	l_2
v_8	B_0	v_{20}	l_2, l_3	v_{18}	B_1	v_{10}	l_2, l_3
v_9	B_0	v_{21}	l_2, l_3	v_{21}	B_1	v_9	l_2, l_3

(d)

super node v_H	v_H's type	*Info*
v_{H1}	*star*	$v_{H1.c} = v_2$
v_{H2}	*CBV*	$v_{H2.vset} = \{v_{12}, v_{13}, v_{18}, v_{21}\}$
v_{H3}	*clique*	
v_{H4}	*path*	$v_{H4.list} = \langle v_4, v_{23}, v_{24}, v_{25} \rangle$
v_{H5}	*singleton*	

Fig. 5: (a) A small fraction of G^k, G^k_{small}; (b) outsourced/contracted graph $G^{out} = f_C(G^k_{small}) = (V_H, E_H, f'_C)$; (c) mapping function f_A; and (d) synopses $\mathcal{S}$

Algorithm 2 FRESHSUBA

Input: Outsourced graph G^{out}, a single query graph Q, f_A, f_C.
Output: A set of embeddings $\mathcal{M}$ of Q in G^{out}.
1: $\mathcal{M} \leftarrow \emptyset$, $t_{\mathcal{M}} \leftarrow \emptyset$
2: **for each** $u \in V(Q)$ **do**
3: $\quad C(u) \leftarrow$ FilterCandidate(G^{out}, u)
4: $\pi \leftarrow$ ChooseMatchingOrder(Q, C)
5: SubgraphSearch(G^{out}, f_A, f_C, Q, C, π, $\mathcal{M}$, $t_{\mathcal{M}}$)
6: **return** $\mathcal{M}$

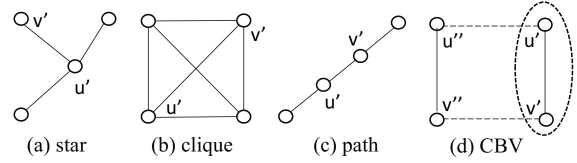


Fig. 6: Edge between u' and v' ($u', v' \in v_H$)

provides sufficient information to either process Q at v_H or bypass v_H, which means that synopses and decontracting superedges without decontracting any topological components are sufficient for answering queries. This guarantees the high query efficiency of FRESH.

A. Subgraph Query

The classical algorithms for subgraph queries [22], [23] can be easily adapted to work with the graph G^{out}. We present the adapted algorithm, called FRESHSUBA, in Algorithm 2. Compared to existing algorithms, FRESHSUBA introduces two significant modifications. The first modification involves checking whether a query vertex is matched by a vertex contained within a supernode in G^{out} (see *FilterCandidate*, Line 3). The second modification is verifying the existence of an edge in G^{out} that matches with a query edge (see *SubgraphSearch*, Line 5). More details are provided below.

FilterCandidate (Line 3). For a query vertex u in Q and a supernode v_H, the reverse function $f'_C(v_H)$ can help to check if v_H contain the label of u. In addition, the degree of each vertex in v_H can be easily derived without decontracting the supernode. For example, if v_H is a star (*i.e.,* v_H.type = star), the degree of the center node is no less than $N - 1$ where N is the number of vertices in the star. For a vertex v in a path, if v is neither source nor target, its degree is 2. If v_H is a CBV (*i.e.,* supernode contracted from one-hop neighbors of vertices in the first block of G^k), we can derive the degree information based on f_A. In particular, we can derive the vertex set from the synopsis of CBV, based on which we can find the symmetric vertices and the corresponding supernode $v_{H'}$ for vertices in CBV. As such, we derive the degree information from $v_{H'}$. Observe that no supernodes or superedges are decontracted.

ChooseMatchingOrder (Line 4). FRESH adopts LDF (*i.e.,* Label and Degree Filtering [22], [23]) to rank query vertices (*i.e.,* π) so that the search space for query matching is relatively small. No adaptation is needed.

SubgraphSearch (Line 5). Since this step involves greedily matching each edge in query Q, it is crucial to ensure that we can easily verify the existence of an edge (u', v') that matches edge $(u, v) \in Q$. When u' and v' are in the same supernode v_H, (1) v_H is a star (*e.g.,* Figure 6(a)), (u', v') exists if u' or v' is the center vertex in v_H; (2) (u', v') always exists if v_H is a clique (*e.g.,* Figure 6(b)); (3) if v_H is a path, (u', v') exists if u' and v' are neighbours (*e.g.,* Figure 6(c)); or (4) if v_H is a CBV and u'' (resp. v'')) is the symmetric node of u' (resp. v') in first block, (u', v') exists if (u'', v'') exists (see Figure 6(d)). When u' and v' are not in the same supernode, it suffices to check if (u', v') exists by decontracting the superedges between the supernode containing u' and that containing v' without decontracting supernodes.

Example 7: Consider the subgraph query of Q (see Figure 1(b)) on G^{out} (Figure 5(b)). By using f'_C, we know that u_2 has candidates v_{H1} and v_{H2}; u_3 has candidate v_{H3}; u_5 has candidate v_{H3} and v_{H2}; u_1 and u_4 both have candidate v_{H4}. Let the query order be u_4, u_1, u_3, u_2, u_5. When SubgraphSearch matches v_{H4} with u_4, it matches v_{H4} with u_1,

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

v_{H3} with u_3, v_{H1} and v_{H2} with u_2, and v_{H3} and v_{H2} with u_5. Consider v_{H4}, since v_{25} connects with other supernodes with superedges, it matches v_{24}, v_{25} with u_4 and u_1, respectively. Then it matches v_6 in v_{H3} with u_3, v_1 in v_{H1} with u_2, and v_7, v_8, v_9 with u_5. As such, there are 3 embeddings for Q. ∎

B. Shortest Distance Query

Diskstra's algorithm [31] is one of the best known algorithms for the shortest distance query. We adapt it to the outsourced graph G^{out} and outline the adapted algorithm, FRESHDIST, in Algorithm 3. The main differences between Diskstra's algorithm and FRESHDIST are two-fold. First, FRESHDIST performs *Edge Relaxation* [31] on edges within a supernode or edges between supernodes (see *ModDijkstra*, Line 2). Second, FRESHDIST needs to handle the case where s and t are in different blocks. FRESHDIST addresses this challenge by decomposing the distance into the distance between internal vertices and the distance between boundary vertices (*e.g.,* v_4 and v_{12}, Figure 2). Formally, the shortest distance between s and t is $d(s,t) = \min_{v_s,v_t} d(s,v_s)+d(v_s,v_t)+d(v_t,t)$ where v_s and v_t are boundary vertices, and v_s (resp. v_t) is in the same block as s (resp. t). The correctness is guaranteed by Theorem 1.

*Definition 4 (**Boundary Vertex**):* Given a k-automorphic graph G^k, if a vertex u in a block B_i has at least one neighbour in another block B_j where $B_i \neq B_j$, u is boundary vertex.

Algorithm 3 FRESHDIST

Input: outsourced graph G^{out}, f_A, f_C and *Dist* query (s,t).
Output: the shortest distance $d(s,t)$.

1: $\mathcal{D}(s,v_s), \mathcal{D}(v_t,t) \leftarrow \{0\}$, $curmin \leftarrow \infty$.
2: $\mathcal{D}(s,v_s) \leftarrow$ ModDijkstra(s, G^{out}, f_C, f_A)
3: $\mathcal{D}(v_t,t) \leftarrow$ ModDijkstra(t, G^{out}, f_C, f_A)
4: **for each** vertex pair (v_s,v_t) s.t. $v_s \in B_s$ and $v_t \in CBV$ **do**
5:     **if** $curmin > d(s,v_s)+d(v_s,v_t)+d(v_t,t)$ **then**
6:         $d(s,t) \leftarrow d(s,v_s)+d(v_s,v_t)+d(v_t,t)$
7:         $curmin \leftarrow d(s,v_s)+d(v_s,v_t)+d(v_t,t)$
8: **return** $d(s,t)$

FRESHDIST. If s and t belong to different blocks, they can be mapped to the same block using the mapping function f_A. Let's assume that $k=2$ for the sake of simplicity. Suppose s is in the first block and t is in the second block, we can map t to the vertex $t' = f_A(t)$ in the first block. To compute the shortest distance $d(s,v_s)$ for each boundary vertex $v_s \in B_s$ (where B_s represents all the boundary vertices in the first block), we utilize the modified Dijkstra algorithm described in Line 2 of Algorithm 3. Similarly, for each boundary vertex $v_t \in CBV$ (in the case of $k=2$, CBV represents the supernode in the second block), we can compute the shortest distance $d(v_t,t) = d(v_t,t')$ (Line 3). By examining all possible vertex pairs v_s and v_t, we derive the shortest distance (Lines 4 to 7).

ModDijkstra. Consider the computation of $d(s,v_s)$. Compared to the traditional Dijkstra's algorithm, the minor modification is the relaxation of edge weight when moving a vertex v_x from unvisited set $V \backslash S$ to the visited set S. Specifically, if v_x and v_y are in the same supernode (denoted as $f_C(v_x) = f_C(v_y) = v_H$), v_x updates $d(v_y)$ using the distance information (*i.e.,* $v_H.dis$) stored in the synopses, where the shortest distance has

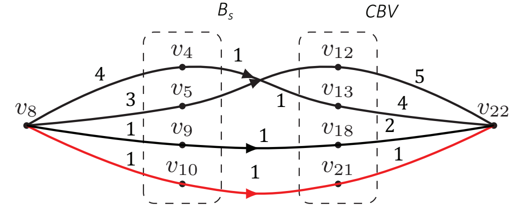


Fig. 7: FRESHDIST

been pre-computed. On the other hand, if $f_C(v_x) \neq f_C(v_y)$, this can be computed by decontracting the superedges between $f_C(v_x)$ and $f_C(v_y)$ using the decontraction function f_D.

Example 8: Consider the shortest distance query $(s = v_8, t = v_{22})$ on G^{out} in Figure 5. Suppose that all edge weights are 1. The boundary vertices B_s are $\{v_9, v_{10}, v_4, v_5\}$ and CBV are $\{v_{12}, v_{13}, v_{18}, v_{21}\}$ (see Figure 7). We first map $t = v_{22}$ to v_{11}. Then, the shortest distances from v_{22} to each vertex in CBV are: $d(v_{22},v_{21}) = 1$, $d(v_{22},v_{18}) = 2$, $d(v_{22},v_{12}) = 6$, $d(v_{22},v_{13}) = 5$. The shortest distances from $s = v_8$ to each vertex in B_s are: $d(v_8,v_9) = 1$, $d(v_8,v_{10}) = 1$, $d(v_8,v_4) = 4$, $d(v_8,v_5) = 3$. In addition, by decontracting v_{H3} and v_{H2}, we derive the shortest distance from v_{10} to v_{21} as 1. As such, the shortest distance from $s = v_8$ to $t = v_{22}$ is $d(v_8,v_{10}) + d(v_{10},v_{21}) + d(v_{22},v_{21}) = 1+1+1 = 3$. ∎

Lemma 2: Let $p(s,t)$ be the shortest path from s to t, u be a vertex along the shortest path, $p(s,u)$ and $p(u,t)$ are the shortest path for (s,u) and (u,t).

Theorem 1: For any vertex pair (s,t), if s and t belong to B_i and B_j, respectively, and $B_i \neq B_j$, there must exist at least one boundary vertex v_s in block B_i and v_t in block B_j such that both v_s and v_t are in the shortest path from s to t.

C. Triangle Counting

We further propose the FRESHTRIC algorithm for triangle counting. Compared to traditional triangle counting [26], [27], FRESHTRIC verifies the existence of an edge that forms a triangle in G^{out}. In particular, FRESHTRIC visits each supernode v_H in G^{out}, if v_H is a CBV (*i.e.,* $v_H.type = CBV$), all triangles in CBV are appended to R_{CV}. If v_H is a clique, all vertices in v_H are appended to R_{CV}. Triangles formed by vertices from different supernodes are appended to R_{Tri}. R_{CV} and R_{Tri} are returned to the client for postprocessing.

VI. POST-PROCESSING AND DISCUSSION

Once the client receives the query results $R^{\mathcal{Q}}$ on G^{out} from a cloud, there is a need to figure out the corresponding results on G, $R^{\mathcal{Q}}_G$. This section discusses in detail how to filter and retrieve the complete results on G for three query classes.

A. Post Processing

*Post Processing for SubA (*PROCESSRESULTSUBA*):* Algorithm 4 outlines post processing process for $SubA$. It takes the original graph G, results $R^{\mathcal{Q}}$ on G^{out} and mapping function f_A as inputs and computes the refined results $R^{\mathcal{Q}}_G$ on G. First, it maps each matched result (*also known as* embedding)

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

$M \in R^{\mathcal{Q}}$ to other unseen blocks in G^k (*i.e.,* blocks other than block 0) (Line 2 to Line 4) based on the mapping function. In particular, for a vertex u (resp. v) in the first block, f_A^i maps it to $f_A^i(u)$ (resp. $f_A^i(v)$) in block $((i+1) \mod k)$. If an edge $(u, v) \in M$, there is an edge $\big(f_A^i(u), f_A^i(v)\big)$ in block $((i+1) \mod k)$. Therefore, we can derive $(k-1)$ embeddings for M, *i.e.,* $M_i = f_A^i(M)$ where $i \in [0, k-1]$. Then, it filters the k embeddings on G^k based on data graph G by removing the embeddings with noise edges (*i.e.,* CheckEdge, Line 5) or labels (*i.e.,* CheckLabel, Line 5), which results in the exact results $R_G^{\mathcal{Q}}$ (Lines 5 to 7).

Algorithm 4 PROCESSRESULTSUBA

```
Input: Original graph G, results R^Q on G^out and mapping function f_A.
Output: refined results R_G^Q on G.
1: R_G^Q ← ∅
2: for each embedding M ∈ R^Q do
3:     for i = 0, 1, ··· , k − 1 do
4:         M_i ← f_A^i(M)
5:         res ← CheckLabel(G, M_i) ∧ CheckEdge(G, M_i)
6:         if res is true then
7:             R_G^Q ← R_G^Q ∪ M_i
8: return R_G^Q
```

Post Processing for Dist: Recall that each noise edge (u, v) is assigned with the weight $w(u, v) = min_{t \in N(u,v)}(w(u, t) + w(t, v)) + \epsilon$ where $N(u, v)$ is the common neighbors of u and v, and ϵ is drawn from a Laplace distribution (see Section IV-D). There is no need to post-process the results from the cloud, since these noise edges will not exist in the shortest paths and have no effect on the final shortest distance.

Post Processing for Tric: As will be seen in Section V, the cloud side returns two kinds of results to the client side, *i.e.,* $R_{CV} \subseteq R^{\mathcal{Q}}$ and $R_{Tri} \subseteq R^{\mathcal{Q}}$ where R_{CV} is a set of a vertex set in a clique and R_{Tri} is a set of triangles in the CBV or crossing at least two supernodes. Let $Count_1$ be the number of triangles crossing supernodes, and $Count_2$ be the number of triangles in supernodes, the total number of triangles is $Count_1 + Count_2$. We calculate $Count_1$ and $Count_2$ below.

(1) Computing $Count_1$: for each triangle in R_{Tri}, if it contains noise edges, it is removed from R_{Tri}. Finally, $Count_1$ is the number of remaining triangles in R_{Tri}.

(2) Computing $Count_2$: suppose there are $|R_{CV}|$ vertex sets in R_{CV}, and each vertex set $R_{CV}(i)$ is a set of vertices in the same clique that may contain noise edges. Let $Count_2(i) = |R_{CV}(i)|$ be the number of vertices in the clique $R_{CV}(i)$, if it contains no noise edge, there are $\binom{|R_{CV}(i)|}{3}$ triangles. Otherwise, to facilitate the calculation of $Count_2(i)$, each edge in the clique is assigned an invisible weight of $|R_{CV}(i)| - 2$. Note that the invisible weight is used to indicate how many triangles the associated edge appears in. Then, for each edge in the clique, if it is a noise edge, it will be removed, and its neighbors' invisible weight will be decreased by 1. In addition, $Count_2(i)$ decreases by the invisible weight of the deleted edge. Finally, $Count_2$ is $\sum_{i \in [1, |R_{CV}|]} Count_2(i)$.

Lemma 3: The time complexity of PROCESSRESULTSUBA *is $O(k|\mathcal{M}||\overline{Q}|)$ where $\mathcal{M}$ is the embedding set from cloud side and $|\overline{Q}|$ denotes the average size for the query graph Q. The time complexity of* PROCESSRESULTTRIC *is $O(|R_{Tri}| + \sum_{i=1}^{|R_{CV}|} |E'_{CV}(i)||R_{CV}(i)|)$ where $|E'_{CV}(i)|$ denotes the number of noise edges in clique i.*

B. Analysis of Complexity, Accuracy, and Compression Ratio

Complexity. Recall that for a graph G, a k-automorphic graph is G^k; vertices in the first block of G^k is V_{B_0}; one-hop neighbors of vertices in V_{B_0} is V_{N_0}; $G^k_{small} = (V_{small}, E_{small})$ where $V_{small} = V_{B_0} \cup V_{N_0}$, $E_{small} \subseteq V_{small} \times V_{small}$; $G^{out} = f_C(G^k_{small})$ is an outsourced graph.

Lemma 4: The time complexities of FRESH, AUT, SUC, and CAU for a subgraph query q with size $V(q)$ are $\mathcal{O}\big((T(v_H)|V(G^{out})|)^{|V(q)|}\big)$, $\mathcal{O}\big(|V(G^k)|^{|V(q)|}\big)$, $\mathcal{O}\big(|V(G^k_{small})|^{|V(q)|}\big)$, $\mathcal{O}\big((\frac{T(v_H)|V(G^k)|}{C})^{|V(q)|}\big)$, respectively, where $T(v_H)$ is the time taken to match a query vertex over a supernode v_H; $|V(G^{out})|$ is the number of supdernodes in G^{out}; $|V(G^k)|$ (resp. $|V(G^k_{small})|$) is number of vertices in G^k (resp. $V(G^k_{small})$); C is the average size of a supernode. In addition, $\mathcal{O}\big((T(v_H)|V(G^{out})|)^{|V(q)|}\big) \leq \mathcal{O}\big(|V(G^k_{small})|^{|V(q)|}\big) \leq \mathcal{O}\big(|V(G^k)|^{|V(q)|}\big)$ and $\mathcal{O}\big((T(v_H)|V(G^{out})|)^{|V(q)|}\big) \leq \mathcal{O}\big((\frac{T(v_H)|V(G^k)|}{C})^{|V(q)|}\big)$.

Lemma 5: The time complexities of FRESH, AUT, SUC, and CAU for triangle counting are $\mathcal{O}\big((|V(G^k_{small})| - C_{star} - 2C_{path})^3\big)$, $\mathcal{O}\big(|V(G^k)|^3\big)$, $\mathcal{O}\big(|V(G^k_{small})|^3\big)$, $\mathcal{O}\big((|V(G^k)| - kC_{star} - 2kC_{path})^3\big)$, respectively, where C_{star} (resp. C_{path}) is the number of stars (resp. paths) in G^{out}. $\mathcal{O}\big((|V(G^k_{small})| - C_{star} - 2C_{path})^3\big) \leq \mathcal{O}\big(|V(G^k_{small})|^3\big) \leq \mathcal{O}\big(|V(G^k)|^3\big)$ and $\mathcal{O}\big((|V(G^k_{small})| - C_{star} - 2C_{path})^3\big) \leq \mathcal{O}\big((|V(G^k)| - kC_{star} - 2kC_{path})^3\big)$.

Lemma 6: The time complexities of FRESH, AUT, SUC, and CAU for the shortest distance query are $\mathcal{O}\big((T(v_H)|V(G^{out})| + |V(G^k_{small})|)\log(|V(G^k_{small})|)\big)$, $\mathcal{O}\big((|E(G^k)| + |V(G^k)|)\log(|V(G^k)|)\big)$, $\mathcal{O}\big((|E(G^k_{small})| + |V(G^k_{small})|)\log(|V(G^k_{small})|)\big)$, $\mathcal{O}\big((\frac{T(v_H)|V(G^k)|}{C} + |V(G^k)|)\log(|V(G^k)|)\big)$, respectively, where $T(v_H)$ is the times required to relax the distance of edges in a supernode v_H; $|V(G^{out})|$ is the number of supdernodes in G^{out}; $|V(G^k)|$ (resp. $|V(G^k_{small})|$) is the number of vertices in G^k (resp. $V(G^k_{small})$); C is the average size of a supernode. In addition, $\mathcal{O}\big((T(v_H)|V(G^{out})| + |V(G^k_{small})|)\log(|V(G^k_{small})|)\big) \leq \mathcal{O}\big((|E(G^k_{small})| + |V(G^k_{small})|)\log(|V(G^k_{small})|)\big) \leq \mathcal{O}\big((|E(G^k)| + |V(G^k)|)\log(|V(G^k)|)\big)$ and $\mathcal{O}\big((T(v_H)|V(G^{out})| + |V(G^k_{small})|)\log(|V(G^k_{small})|)\big) \leq \mathcal{O}\big((\frac{T(v_H)|V(G^k)|}{C} + |V(G^k)|)\log(|V(G^k)|)\big)$.

Accuracy. The correctness of the query is guaranteed. We establish the correctness of triangle counting (or subgraph query) by demonstrating that the FRESHTRIC (or FRESHSUBA) algorithm can deliver all potential results, while the accompanying post-processing procedure effectively filters out false positives.

Theorem 2: The FRESH algorithm guarantees the accuracy of the triangle-counting query.

Theorem 3: The FRESH algorithm guarantees the accuracy of the subgraph query.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

The correctness of the shortest distance query is also guaranteed. We demonstrate this by showing that the FRESHDIST algorithm can exactly find the shortest distance.

Theorem 4: The FRESH algorithm guarantees the accuracy of the shortest distance query.

Compression Ratio. Given the importance of G^k in defending against structural attacks, the compression ratio is defined as the size of G^k to the size of G^{out}.

Lemma 7: Let V be the vertex set of $V(G^k)$, and $C_{star}, C_{clique}, C_{path}, C_{sin}$ be the number of stars, cliques, paths, and singletons in G^{out}, respectively, the compression ratio is $|V|/(C_{star}+C_{clique}+C_{path}+C_{sin}+k-1)$, which is bounded by $k|V|/(|V|+(k-1)k)$.

VII. Optimizations

Observe that the number of noise labels introduced in the process of building G^k has an impact on the labeled query $SubA$, to efficiently process such queries, we design a novel optimization to further boost query efficiency. We begin with the estimation for the search space of $SubA$.

A. Search space of subgraph query

Consider k-automorphic graph G^k with $|L|$ types of labels. Let $F_{G^k}(i)$ (resp. $F_Q(i)$) be the probability a vertex in G^k (resp. Q) with label i, D_{G^k} the average degree in G^k. The search space for SubA query can be estimated as follows [2].

$$\frac{|V_{G^k}|}{k}\Big[\sum_{i=1}^{|L|}F_{G^k}(i)F_Q(i)\Big]\Big[D_{G^k}\sum_{i=1}^{|L|}F_{G^k}(i)F_Q(i)\Big]^{D_{G^k}} \quad (1)$$

where $F_{G^k}(i)=\frac{|V_{G^k,i}|}{|V_{G^k}|}, F_Q(i)=\frac{|V_{Q,i}|}{|V_Q|}$. Therefore, we shall reduce the number of (noise) labels (*i.e.,* $|V_{G^k,i}|$) and average degree (D_{G^k}) to reduce the search space.

B. Optimization method

To this end, we proposed the optimized K-Match algorithm in Algorithm 5. It takes data graph G and anonymization parameter k as input and generates k-automorphic graph G^k. First, it follows the existing K-Match algorithm to build AVT (Line 2). In the AVT, vertices with similar degrees may be put into the same row of AVT. However, vertices with similar degrees may have different labels, resulting in the vertex in the first block B_0 of G^k assigned with labels of additional vertices in the same row (*i.e.,* vertices in B_1 to B_{k-1}). Then, it creates a BFS spanning tree to address this problem (Line 3). In particular, for any two vertices, if their symmetric vertices in block B_0 lie in the same depth of the spanning tree, it adjusts the AVT by swapping positions of these vertices in AVT if the swapping can decrease the amount of labels by at least 2 (ADJUSTAVT, Line 4). Finally, edge alignment is performed to derive the final G^k (Line 5).

Theorem 5: Each vertex in AVT is swapped at most once during ADJUSTAVT.

Remark. A keen reader may wonder why designing the aforementioned optimization instead of adopting state-of-the-art indexing methods (*e.g.,* 2-hop-labeling and CT-index [33], [71]) to boost the query efficiency. The prior work typically targets a specific class of queries, *e.g.,* 2-hop labeling [33] is for the shortest distance/path queries. However, in practice, multiple applications frequently operate on the same graph simultaneously. It is impractical to switch between different applications. In addition, building indices for each query class in use would be prohibitively expensive [18].

Algorithm 5 K-MatchOPT

Input: Original graph G, parameter K.
Output: Anonymized network G^k.
1: $(B_0, B_1, \cdots, B_{k-1}) \leftarrow$ GRAPHPARTITION(G, K)
2: AVT $\leftarrow$ CREATEAVT($B_0, B_1, \cdots, B_{k-1}$)
3: $\mathcal{A} \leftarrow$ BUILDSPANNINGTREE(B_0)
4: AVT $\leftarrow$ ADJUSTAVT(($B_0, B_1, \cdots, B_{k-1}$), $\mathcal{A}$, G, AVT)
5: $G^k \leftarrow$ EDGEALIGNMENT(AVT, G)
6: **return** G^k

TABLE II: List of datasets.

Dataset	Number of vertices $\lvert V_G\rvert$	Number of edges $\lvert E_G\rvert$	Number of labels $\lvert l_G\rvert$
DBLP	317K	865K	100
NotreDame	325K	1M	200
Amazon	334K	925K	100
BerkStan	685K	7.6M	1200
Google	875K	5.1M	1200
RoadNet	1M	3M	2000
YouTube	1.1M	2.9M	46
USA	1.2M	2.8M	2000
Livejournal	4M	35M	2000
uk-2002	18M	0.261 Billion	6000

VIII. Performance Study

FRESH is implemented in C++11. The client is a laptop with M1 CPU and 16GB RAM running MacOS 11. The cloud server provided by Alibaba Cloud is a Linux Server with 8 CPU cores and 128GB RAM.

A. Experimental setup

Datasets: (a) Web graphs: BerkStan, Google, NotreDame and uk-2002; (b) Community networks: DBLP, Amazon, YouTube and Livejournal; and (c) Road networks: RoadNet and USA. The statistics are in Table II.

Competitors: We compare FRESH against baselines, *i.e.,* AUT, SUC, and CAU (see Section III-C). Note that they adopt the same query processing methods as FRESH.

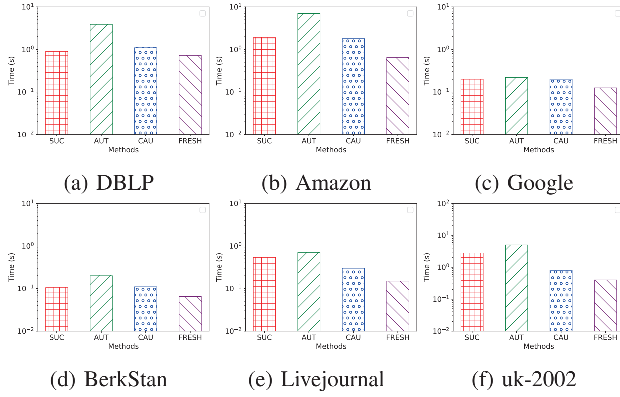
Performance metrics and Parameter settings: We measure query processing time, pre-computing time (*i.e.,* contraction time and anonymization time), contraction percentage $\eta=(|V(G^{out})|+|E(G^{out})|)/(|V(G)|+|E(G)|)$ (*i.e.,* the percentage of the size of G^{out} to that of G), and space storage cost. Unless otherwise stated, $k=2$, $c_l=4$, $c_u=200$.

B. Experimental Results

As result filtering in the client side is straightforward and lightweight, we mainly present the performance of FRESH on cloud side on three types of queries: *SubA*, *Dist* and *TriC*.

Exp 1: FRESH vs. Baselines. To compare FRESH and baselines for *SubA*, we adopt a random walk on the original data graph G to generate 200 query graphs and report the average processing time in Figure 8. Observe that FRESH outperforms baselines while AUT performs the worst among all baselines. The main reason lies in that FRESH can greatly reduce the

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.



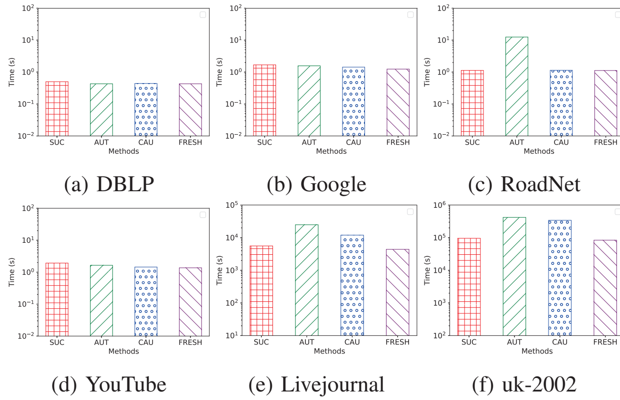
(a) DBLP (b) Amazon (c) Google

(d) BerkStan (e) Livejournal (f) uk-2002

Fig. 8: Baseline comparison, *SubA*

TABLE III: Storage Cost (MB)

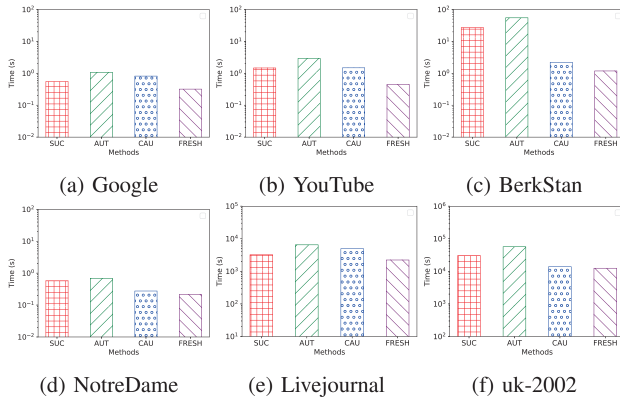
Dataset	G	G^k	G^k_{small}	G^k_C	G^{out}
Amazon	19	31	16	16	7.6
DBLP	18	28	15	15	6.7
RoadNet	41	52	29	26	13
BerkStan	118	197	86	45	21
Google	82	133	65	45	21
YouTube	61	100	54	60	27
Livejournal	479	996	515	537	285
uk-2002	4812	8499	3891	1433	783



(a) DBLP (b) Google (c) RoadNet

(d) YouTube (e) Livejournal (f) uk-2002

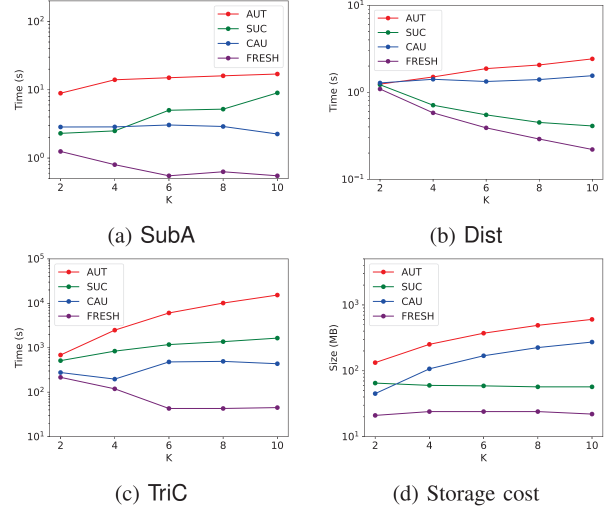
Fig. 9: Baseline comparison, *Dist*



(a) Google (b) YouTube (c) BerkStan

(d) NotreDame (e) Livejournal (f) uk-2002

Fig. 10: Baseline comparison, *TriC*



(a) SubA (b) Dist

(c) TriC (d) Storage cost

Fig. 11: Effect of k, query time and storage cost

graph size (see storage cost of G^{out}, Table III) and the compact graph further boosts the query processing of *SubA*. In contrast, AUT is performed on G^k, which incurs the largest space storage cost (third column, Table III) and takes the longest time. As shown in Figure 8(e) and (f), the performance improvement on larger graphs is more considerable. This justifies the significance of FRESH for *SubA*.

Next, we compare FRESH and baselines for *Dist* with 1000 queries and plot the results in Figure 9. In general, FRESH performs the best compared to baselines. Although this advantage may not be particularly evident in some datasets (*e.g.,* DBLP, Figure Figure 9(a)), this is because the comparison was made when $k = 2$. When $k = 2$, the size difference between G^k or G and G^{out} is not very significant in some datasets. As will be seen later (see Exp 2), this advantage will gradually become more pronounced when k increases.

We further compare FRESH and baselines for *TriC* and report the results in Figure 10. We can find that FRESH outperforms all the baselines on all tested datasets. Compared to SUC, CAU, and FRESH, AUT takes the longest processing time as it greatly enlarges the graph size to achieve symmetric structures. In conclusion, FRESH outperforms the competitors on all three queries (*i.e.,* $SubA$, $Dist$, and $TriC$) in terms of both query processing time and storage cost.

Exp 2: Effect of k. We vary the value of k from 2 to 10 and compare their performances in $SubA$, $Dist$, and $TriC$. As reported in Figure 11(a), subgraph query processing time of both AUT and SUC increases with the value of k. For AUT, this is apparent since the graph size and search space for $SubA$ increase with k; for SUC, the main reason is that as k increases, although the number of vertices/edges in block 0 may decrease, the corresponding CBV will increase in terms of the number of vertices. Although the total number of vertices and edges does not change significantly, we need to consider the potential edges in CBV in subgraph matching

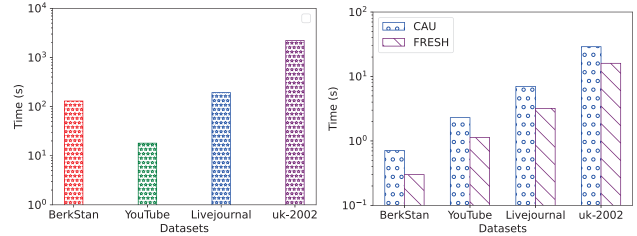
Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

because we need to map the vertices in the CBV to block 0 during the search. As k increases, the number of vertices/edges in CBV increases, as well as the noise edges, which leads to an increase in the search space and query processing time. While the processing time of CAU and FRESH slightly decreases with k. For CAU, as k increases, the number of edges in G^k increases, but the proportion of cliques in G^k also increases. But graph contraction can effectively reduce the impact of edge increase, resulting in a slightly decreasing trend overall. For FRESH, the number of vertices in block 0 decreases as k increases. Although the CBV increases in terms of the number of vertices, the information in CBV is stored in the synopses, which can accelerate the query processing process.

As illustrated in Figure 11(b), as k increases, distance query processing time of AUT and CAU increase while that of SUC and FRESH decrease. The reasons are as follows: as k increases, (1) for CAU, the graph becomes denser, more cliques are formed, and more superedges need to be contracted for $Dist$, resulting in an overall increase in processing time; (2) for SUC, number of vertices in block 0 and its corresponding $Dist$ query cost decrease, although number of vertices in CBV increases, this does not lead to the increase of query cost since shortest distance in CBV are recorded; (3) both the decreases of number of vertices in block 0 and graph contraction lead to the decrease of processing time of FRESH on $Dist$. As shown in Figure 11(c), for $TriC$, the query time of AUT, SUC and CAU increases with k, as graph becomes denser as the k increases. We also report the storage cost in Figure 11(d). As k increases, the size of G^k increases, so the storage cost of CAU and AUT also increases. The size of G^k_{small} does not change significantly overall, although the number of vertices in CBV increases while the number of vertices in Block 0 decreases. Therefore, the storage cost of SUC and FRESH does not change significantly. Compared to SUC, FRESH requires less storage cost.

Exp 3: Processing Time of Graph Contraction. Next, we present the running time of k-automorphic graph construction in Figure 12(a) and graph contraction in Figure 12(b). It is worth noting that although larger graph sizes result in longer running times for k-automorphic graph construction, even the largest graph, uk-2002, can be constructed within 35 minutes. Importantly, this step is performed only once before sending the graph to the cloud side. As depicted in Figure 12(b), FRESH requires less contraction time compared to CAU, which contracts the entire G^k. This demonstrates the efficiency of FRESH in reducing the time required for graph contraction.

Exp 4: Communication Cost. We also included an analysis of the communication cost involved in sending outsourced graphs to the cloud. Table IV presents the communication cost, which increases as the size of the graph grows. However, due to the substantial reduction in the size of G^k achieved by outsourcing G^{out} in FRESH, the communication cost remains within acceptable limits. Even for the largest dataset, it takes less than 10 minutes to transmit G^{out} to the cloud.



(a) Anonymization time (b) Contraction time

Fig. 12: Processing Time of Graph Contraction

TABLE IV: Communication Cost (s)

Dataset	G	G^k	G^k_{small}	G^k_C	G^{out}
Amazon	18	19	15	15	6
DBLP	17	26	13	16	7
RoadNet	31	41	21	20	10
BerkStan	84	136	58	28	15
Google	61	98	44	26	15
YouTube	42	73	36	42	18
Livejournal	319	664	325	368	192
uk-2002	3108	5766	2594	925	542

IX. Conclusion

In this paper, we present FRESH, a versatile framework that efficiently handles multiple classes of graph queries in a single outsourced graph. We introduce a novel graph contraction scheme to reduce a large graph into a compact one while maintaining graph privacy, thereby reducing the size of the outsourced graph. To demonstrate the feasibility of FRESH, we adapt classical graph query algorithms, such as subgraph query, triangle counting, and shortest distance query, to the same compact graph as a proof of concept. Additionally, we implement optimizations to improve query processing efficiency. Our comprehensive experimental results demonstrate that FRESH outperforms traditional techniques. We will explore other graph queries such as community search [72], graphlet counting [73], reachability queries [74], and graph pattern mining [28], [29] on outsourced graphs.

X. Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No: 92270123 and 62372122), the Research Grants Council, Hong Kong SAR, China (Grant No: 15203120, 15226221, 15225921, 15209922, 15208923 and 15210023), and General Research Grants (Grant No: FRG-24-027-FIE) of the MUST Faculty Research Grants (FRG). The research work described in this paper was partially conducted in the JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust, HKUST-China Unicom Joint Lab on Smart Society, and HKUST-HKPC Joint Lab on Industrial AI and Robotics Research.

References

[1] Technical Report. Available at: https://github.com/TechReport2022/FRESH/blob/main/FRESH-Report.pdf.

[2] Chang Z, Zou L, Li F. Privacy preserving subgraph matching on large graphs in cloud. *SIGMOD*, 2016.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.

[3] Hu H, Xu J, Chen Q, et al. Authenticating location-based services without compromising location privacy. *SIGMOD*, 2012.
[4] Huang K, Hu H, Zhou S, et al. Privacy and efficiency guaranteed social subgraph matching. *The VLDB Journal*, 2022.
[5] Zou L, Chen L, Özsu M T. K-automorphism: A general framework for privacy preserving network publication. *PVLDB*, 2009.
[6] Cheng J, Fu A W, Liu J. K-isomorphism: privacy preserving network publication against structural attacks. *SIGMOD*, 2010.
[7] Bron, C and Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 1973.
[8] Eppstein D, Löffler M and Strash D. Listing all maximal cliques in large sparse real-world graphs. *JEA*, 2013.
[9] Wu W, Xiao Y, Wang W, et al. K-symmetry model for identity anonymization in social networks. *EDBT*, 2010.
[10] Sweeney L. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2002.
[11] Machanavajjhala A, Kifer D, Gehrke J, et al. l-diversity: Privacy beyond k-anonymity. *ICDE*, 2006.
[12] Li N, Li T, Venkatasubramanian S. t-closeness: Privacy beyond k-anonymity and l-diversity. *ICDE*, 2007.
[13] Zhou B and Pei J. Preserving privacy in social networks against neighborhood attacks. *ICDE*, 2008.
[14] Hay M, Miklau G, et al. Resisting structural re-identification in anonymized social networks. *PVLDB*, 2008.
[15] Liu K, Terzi E. Towards identity anonymization on graphs. *SIGMOD*, 2008.
[16] G Karypis, V Kumar. Analysis of multilevel graph partitioning. *SC*, 1995
[17] Dominguez-Sal D, et al. A discussion on the design of graph database benchmarks. *Technology Conference on Performance Evaluation and Benchmarking*, 2010.
[18] Fan W, Li Y, Liu M, et al. Making graphs compact by lossless contraction. *SIGMOD*, 2021.
[19] W Han, J Lee, J Lee, et al. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. *SIGMOD*, 2013.
[20] F Bi, L Chang, X Lin, L Qin, W Zhang, et al. Efficient Subgraph Matching by Postponing Cartesian Products. *SIGMOD*, 2016.
[21] J Lee, W Han, R Kasperovics, J Lee, et al. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *VLDB*, 2012.
[22] S Sun, Q Luo, et al. In-Memory Subgraph Matching: An In-depth Study. *SIGMOD*, 2020.
[23] Li Z, Yan J, W Lu, L Zou, et al. Deep Analysis on Subgraph Isomorphism. *arXiv preprint arXiv:2012.06802*, 2020.
[24] M Han, H Kim, G Gi, K Park, W Han, et al. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. *SIGMOD*, 2019.
[25] V Bonnici, R Giugno, et al. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 2013.
[26] X Hu, Y Tao, et al. Massive graph triangulation. *SIGMOD*, 2013.
[27] C Jonathan, et al. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report*, 2008.
[28] Huang K, Hu H, Ye Q, et al. TED: Towards Discovering Top-k Edge-Diversified Patterns in a Graph Database. *SIGMOD*, 2023.
[29] K. Huang, et al. TED$^+$: Towards Discovering Top-k Edge-Diversified Patterns in a Graph Database. *TKDE*. doi: 10.1109/TKDE.2023.3312566
[30] Takuya Akiba, et al. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. *SIGMOD*, 2013.
[31] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1959.
[32] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A Search Meets Graph Theory. *SODA*, 2005.
[33] D Ouyang, L Qin, at al. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. *SIGMOD*, 2018.
[34] M. Jiang, A. W. Fu, et al. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 2014.
[35] Das, Sudipto and Eğecioğlu, Ömer and El Abbadi, Amr. Anonymizing weighted social network graphs. *ICDE*, 2010.
[36] J. Gao, J. X. Yu, R. Jin, J. Zhou, T. Wang, and D. Yang. Neighborhood-privacy protected shortest distance computing in cloud. *SIGMOD*, 2011.
[37] N. Cao, Z. Yang, et al. Privacy-preserving query over encrypted graph-structured data in cloud computing. *ICDCS*, 2011.
[38] L Xu, B Choi, Y Peng, J Xu, S S Bhowmick. A Framework for Privacy Preserving Localized Graph Pattern Query Processing. *SIGMOD*, 2023.
[39] Z Fan, Y Peng, B Choi. Towards efficient authenticated subgraph query service in outsourced graph databases. IEEE Transactions on Services Computing, 2014.
[40] K. Huang, H Liang, et al.," VisualNeo: Bridging the Gap between Visual Query Interfaces and Graph Query Engines. *PVLDB*, 2023.
[41] Huang K, Ye Q, , et al. VINCENT: Towards Efficient Exploratory Subgraph Search in Graph Databases. *PVLDB*, 2022.
[42] Z Chang, L Zou, F Li, et al. Privacy preserving subgraph matching on large graphs in cloud. *SIGMOD*, 2016.
[43] L Zou, L Chen, M T Özsu, et al. K-automorphism: a general framework for privacy preserving network publication. *VLDB*, 2009.
[44] M. Yuan, L. Chen, S. Y. Philip, and T. Yu. Protecting sensitive labels in social network data anonymization. *TKDE*, 2013.
[45] M. Hay, C. Li, G. Miklau, and D. Jensen. Accurate estimation of the degree distribution of private networks. *ICDM*, 2009.
[46] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. *PVLDB*, 2011.
[47] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. Private release of graph statistics using ladder functions. *SIGMOD*, 2015.
[48] K. Huang, et al., MIDAS: Towards Effective Maintenance of Canned Patterns in Visual Graph Query Interfaces. *SIGMOD*, 2021.
[49] S S Bhowmick, et al., AURORA: Data-driven Construction of Visual Graph Query Interfaces for Graph Databases. *SIGMOD*, 2020.
[50] Q. Ye, H. Hu, M. H. Au, X. Meng, and X. Xiao. LF-GDPR:Graph Metric Estimation with Local Differential Privacy. *TKDE*, 2020.
[51] S. Chen and S. Zhou. Recursive mechanism: Towards node differential privacy and unrestricted joins. *SIGMOD*, 2013.
[52] S. P. Kasiviswanathan, K. Nissim, S. Raskhodnikova, and A. Smith. Analyzing graphs with node differential privacy. *TCC*, 2013.
[53] W. Y. Day, N. Li, and M. Lyu. Publishing graph degree distribution with node differential privacy. *SIGMOD*, 2016.
[54] X. Ding, S. Sheng, S. Zhou, et al. Differentially Private Triangle Counting in Large Graphs. *TKDE*, 2021.
[55] Sara Cohen. Data management for social networking. In *SIGMOD*, 2016.
[56] K. Huang, et al., CATAPULT: Data-driven Selection of Canned Patterns for Efficient Visual Graph Query Formulation. *SIGMOD*, 2019.
[57] K. Huang, et al., PICASSO: Exploratory Search of Connected Subgraph Substructures in Graph Databases. *PVLDB*, 2017.
[58] B Zheng, Y Ma, et al. Reinforcement Learning based Tree Decomposition for Distance Querying in Road Networks. *ICDE*, 2023.
[59] B Zheng, J Wan, et al, Christian S.Jensen. Workload-Aware Shortest Path Distance Querying in Road Networks. *ICDE*, 2022.
[60] Q Ye, H Hu, et al. Stateful Switch: Optimized Time Series Release with Local Differential Privacy. *INFOCOM*, 2023.
[61] Q Qian, et al. Collaborative Sampling for Partial Multi-dimensional Value Collection under Local Differential Privacy. *TIFS*, 2023.
[62] X Sun, Q Ye, et al. Synthesizing Realistic Trajectory Data with Differential Privacy. *TITS*, 2023.
[63] Q Ye, H Hu, et al. PrivKVM*: Revisiting Key-Value Statistics Estimation with Local Differential Privacy. *TDSC*, 2021.
[64] Antonio Maccioni and Daniel J Abadi. Scalable pattern matching over compressed graphs via dedensification. In *SIGKDD*, 2016.
[65] Kristen LeFevre and Evimaria Terzi. GraSS: Graph structure summarization. In *SDM*, 2010.
[66] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph Summarization Methods and Applications: A Survey. *ACM Comput. Surv*, 2018.
[67] Bibek Bhattarai, et al. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*, 2019.
[68] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.
[69] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. *SICOMP*, 2003.
[70] Yongjiang Liang and Peixiang Zhao. Similarity search in graph databases: A multi-layered indexing approach. In *ICDE*, 2017.
[71] Li W, Qiao M, Qin L, et al. Scaling up distance labeling on graphs with core-periphery properties. In *SIGMOD*, 2020.
[72] Fang Y, Huang X, Qin L, et al. A survey of community search over big graphs. *The VLDB Journal*, 2020, 29: 353-392.
[73] Ahmed N K, Neville J, Rossi R A, et al. Efficient graphlet counting for large networks. In *ICDM*, 2015.
[74] Cheng J, Shang Z, Cheng H, et al. Efficient processing of k-hop reachability queries. *The VLDB Journal*, 2020, 23(2): 227-252.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on October 31,2024 at 03:01:57 UTC from IEEE Xplore. Restrictions apply.